UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS
MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE –
EUROPEAN MASTER IN SOFTWARE ENGINEERING

Testing of Android Testing Tools: Development of a Benchmark for the Evaluation

Master Thesis

Tahir Javaid

Madrid, July 2015

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: Testing of Android Testing Tools: Development of a Benchmark for the Evaluation
Thesis no: EMSE-2015-03
July 2015

Author: Tahir Javaid
Graduate Student
Universidad Politécnica de Madrid

Supervisor:

Ricardo Imbert
Associate Professor
Computer Science and Engineering School
Universidad Politécnica de Madrid

Co-supervisor:

Alessandra Gorla
Assistant Research Professor
IMDEA Software institute

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

1. Introduction

With the ever growing trend of smart phones and tablets, Android is becoming more and more popular everyday. With more than one billion active users [i] to date, Android is the leading technology in smart phone arena. In addition to that, Android also runs on Android TV, Android smart watches and cars. Therefore, in recent years, Android applications have become one of the major development sectors in software industry. As of mid 2013, the number of published applications on Google Play had exceeded one million and the cumulative number of downloads was more than 50 billion[ii]. A 2013 survey also revealed that 71% of the mobile application developers work on developing Android applications[iii].

Considering this size of Android applications, it is quite evident that people rely on these applications on a daily basis for the completion of simple tasks like keeping track of weather to rather complex tasks like managing one's bank accounts. Hence, like every other kind of code, Android code also needs to be verified in order to work properly and achieve a certain confidence level.

Because of the gigantic size of the number of applications, it becomes really hard to manually test Android applications specially when it has to be verified for various versions of the OS and also, various device configurations such as different screen sizes and different hardware availability. Hence, recently there has been a lot of work on developing different testing methods for Android applications in Computer Science fraternity. The model of Android attracts researchers because of its open source nature. It makes the whole research model more streamlined when the code for both, application and the platform are readily available to analyze. And hence, there has been a great deal of research in testing and static analysis of Android applications. A great deal of this research has been focused on the input test generation for Android applications. Hence, there are a several testing tools available now, which focus on automatic generation of test cases for Android applications. These tools differ with one another on the basis of their strategies and heuristics used for this generation of test

cases. But there is still very little work done on the comparison of these testing tools and the strategies they use.

Recently, some research work has been carried out[iv] in this regard that compared the performance of various available tools with respect to their respective code coverage, fault detection, ability to work on multiple platforms and their ease of use. It was done, by running these tools on a total of 60 real world Android applications. The results of this research showed that although effective, these strategies being used by the tools, also face limitations and hence, have room for improvement.

The purpose of this thesis is to extend this research into a more specific and attribute-oriented way. Attributes refer to the tasks that can be completed using the Android platform. It can be anything ranging from a basic system call for receiving an SMS to more complex tasks like sending the user to another application from the current one. The idea is to develop a benchmark for Android testing tools, which is based on the performance related to these attributes. This will allow the comparison of these tools with respect to these attributes. For example, if there is an application that plays some audio file, will the testing tool be able to generate a test input that will warrant the execution of this audio file? Using multiple applications using different attributes, it can be visualized that which testing tool is more useful for which kinds of attributes.

In this thesis, it was decided that 9 attributes covering the basic nature of tasks, will be targeted for the assessment of three testing tools. Later this can be done for much more attributes to compare even more testing tools. The aim of this work is to show that this approach is effective and can be used on a much larger scale.

One of the flagship features of this work, which also differentiates it with the previous work, is that the applications used, are all specially made for this research. The reason for doing that is to analyze just that specific attribute in isolation, which the application is focused on, and not allow the tool to get bottlenecked by something trivial, which is not the main attribute under testing. This means 9 applications, each focused on one specific attribute.

The main contributions of this thesis are:

- A summary of the three existing testing tools and their respective techniques for automatic test input generation of Android Applications.
- A detailed study of the usage of these testing tools using the 9 applications specially designed and developed for this study.
- The analysis of the obtained results of the study carried out. And a comparison of the performance of the selected tools.

Future Extensions: With successful results, this research can be continued further to a much higher level. The complete benchmark of applications can be developed with one application each for every possible attribute of Android API. And all these benchmark applications tested across all the testing tools available in the market. With the development of this benchmark, the testing for Android can be revolutionized and be made very easy. Any development entity can test their application with the required testing tools depending on the attributes their application uses. All they will be required to do is to follow the benchmark and select the appropriate testing tools to test their application. Not only will this save time and resources, but also increase the efficiency of the testing process, which is one of the most important phases of the development process of software these days. The process of Quality Assurance will be improved with a prominent difference.

The outline of this thesis is as follows:

2. Background.
3. AndroTest Execution Environment.
4. Overview of Testing Tools
5. Decided Tools
6. Benchmark Applications
7. Evaluations
8. Conclusion
9. References

2. Background

Started in 2013, Android's potential was soon discovered by the big fishes of the computer software industry. Hence, acquired by Google in 2005, it became the flagship platform for smart phone technology in no time. Due to this rapid growth and speedy nature of the whole smart phone application industry, the majority of the development companies focused on the fast production of applications and their deployment on the App Store to generate profits. Hence, there was very little research on the topics like software verification and validation and automated testing until recently. As mentioned earlier, now there are some testing tools available that are focused on android input generation and hence, testing. One might argue here that since Android applications mostly use Java code, why can the already existing testing tools for Java not be used for the testing purposes of Android. This section of the thesis will analyze the overall Android application structure and show why is it necessary to build separate tools with separate strategies to undertake the task of Android testing. This information is also useful for the later sections of the thesis, which deal with rather technical aspects of this research.

It is common knowledge that Android applications are written in Java. When built, this source code is compiled into Java bytecode and then further converted into Dalvik bytecode. Then this bytecode is stored in *.dex* files, which are machine executable files. A collection of these files along with the application resources, are then compressed into *.apk* files, which eventually are installed on the targeted machines.

As shown in the Figure 1[v], there are three software layers running below the actual Android applications.

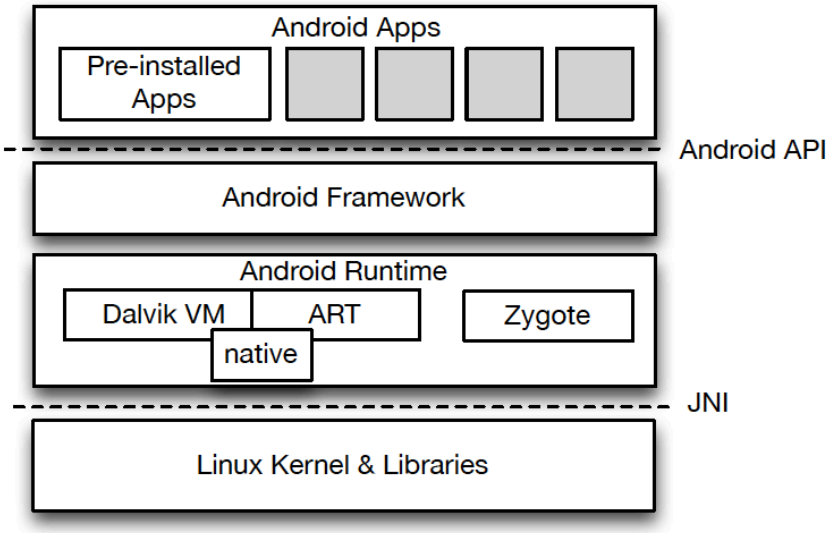


Figure 1: Layered Representation of Android Platform

The first layer, The Android Platform, is basically the Android API on which the application is built and this layer bridges the application to the lower levels of the Operating system. This API is a rapid evolving technology. With 21 releases till now, this technology is quite evolved as compared to its starting releases. This is the origination of the famous problem of Fragmentation in Android platform. Tackling Fragmentation is one of the most common and important attributes of Android application development. Due to the rapid releases of the new APIs, a huge number of devices run an older version of the API at any given time. Hence, developers aim to develop their app to support the most of devices. This affects testing as well since it produces the need to test the application for not only different APIs, but also different hardware.

The second layer, The Android Runtime, all the applications are managed by separate virtual machines. Since the bytecode used is Dalvik, the virtual machines are also Dalvik virtual machines.

And finally under all this is the third layer, which is basically a Linux Kernal that deals with the functionality of the Operating system.

Application Structure

Android applications have a set of components that are dealt with separate classes using Object Oriented Approach. Other than the implementation of these components, they are also defined in a mandatory file named *AndroidMenifest.xml*. These components can either be activities, services, receivers or providers.

Activities are the most common component of an Android application. They are basically the UI part of the application. One activity represents one screen layout. Activities can be designed to listen to system events and also contain UI building blocks like text fields, buttons etc. Normally every activity has another XML file to define the design of the activity. Every activity has a lifecylcle, which is depicted in Figure 2[vi].

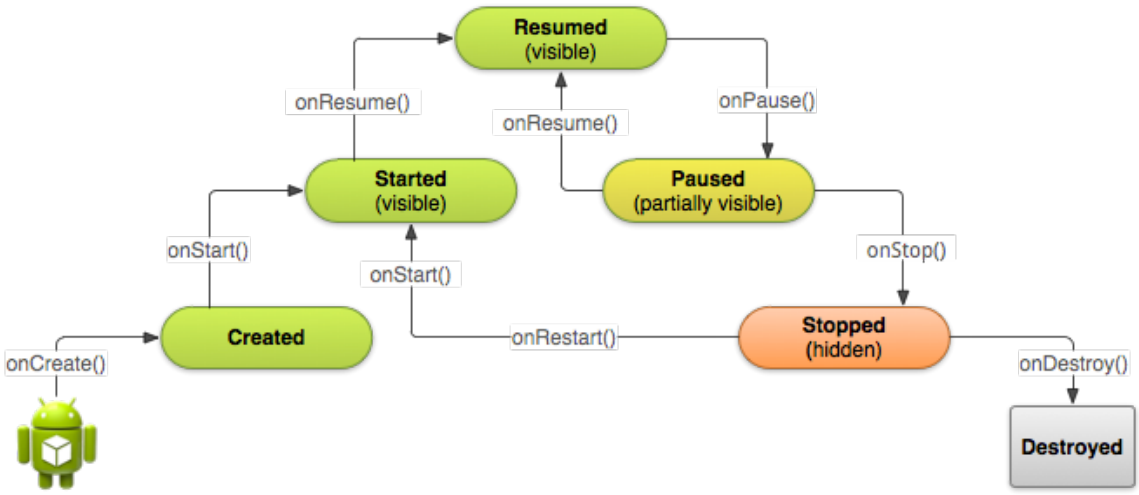


Figure 2: Activity Lifecycle

Services can be anything that is running in the background and as evident from the name, providing some service. Further in the thesis, it will be discussed how these services are mostly ignored by the testing tools due to the lack of a UI element.

Receivers are basically listeners. Like in Java, applications register to listen to an event, receivers provide the similar job in Android. Receivers are very important for handling the communication between system events and the application. For instance, an

application can register for the change in connectivity of the phone to any type of network. Then, whenever the phone's connectivity status is changed, lets say it switches from Cellular Data to Wifi, this receiver will receive a system call.

Providers, also known as Content Providers, provide the access to local content/data. For instance, an application that needs to access the contacts information of the user, needs to implement a content provider.

Other than the definition of these components, AndroidManifest contains the permission information, coverage tools information, android sdk target and minimum versions, and a lot of other information.

Above information and years of experience suggest that despite being written in Java, Android applications face different problems than normal Java applications. The nature of the whole platform is different. And Android applications face a very different kind of faults[vii]. That is why there has been plenty of research recently on developing targeted testing tools for Android.

3. AndroTest Execution Environment

This section provides a detailed explanation of the AndroTest execution environment, the platform used to undertake the empirical experimentation for this research. Since this research is a continuation of the a previous research, this platform was established by the people who undertook the previous research. But it required certain configurations for the current use. The details of all the major characteristics of this environment along with the changes are covered in this section.

The AndroTest environment basically works as a virtual machine configured to run Linux. The setup uses Virtual Box[viii] along with Vagrant[ix] to run this virtual machine locally on any system. Once these tools are installed and configured, the virtual machine can be accessed through command line via an SSH connection. The details for setting up this environment can be found on the following web address (http://bear.cc.gatech.edu/~shauvik/androtest/). A total of ten virtual machines are

configured (run1-run10) for the purpose of dividing the workload through parallel processing. Typing the 'vagrant up' command with the name of the machine as arguments can start each one of these machines. Once the machine is running, the 'vagrant ssh' command can be used to get the access of the machine. For instance, to start the machine run1, the following has to be done.

- ➔ vagrant up run1
- ➔ vagrant ssh run1

Once the SSH connection is formed, the user is redirected to the root directory of the virtual machine which contains all the basic folders required for running the testing tools. A snapshot of the root directory is shown in Figure 3. As shown in the figure, there are the following folders available in the root directory:

1. Android_apps_benchmark
2. Android-ndk-r10
3. Android-sdk-Linux
4. Apktool
5. Lib
6. Scripts
7. Subjects
8. Tools

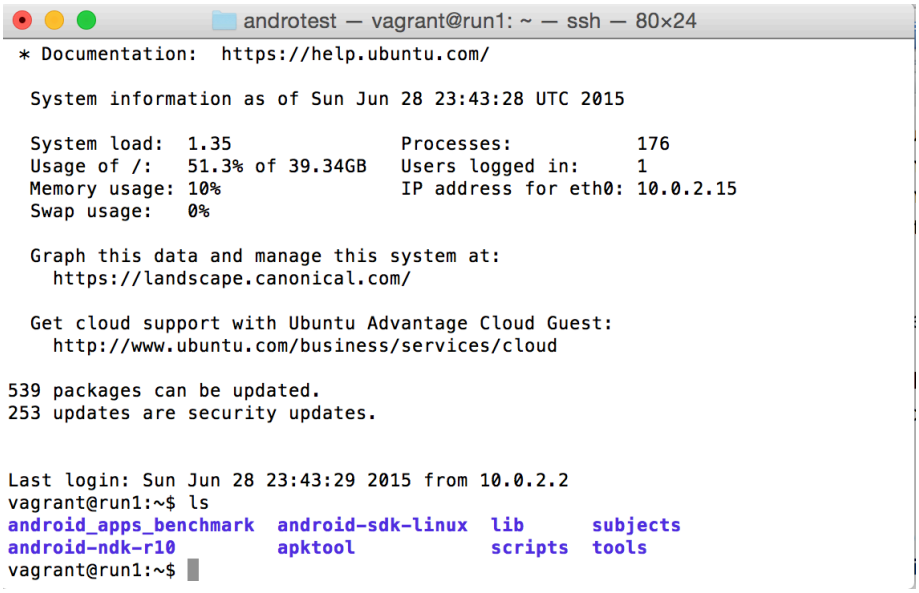


Figure 3: Root directory of Virtual Machine run1

As few of these folders are quite clear from the name as to what they contain, here is a brief detail for the others.

Folder 'Android_apps_benchmark', is originally not present in the virtual machine. It was added for this research to transfer all the benchmark apps developed for this study. To ensure a smooth flow, a GIT repository was created to enforce a version control system. All the developed applications were kept through this version control. The repository was cloned into this folder to transfer the applications to the virtual machine. Since this version control was not already incorporated in the environment prior to this, it was also configured for this research.

The 'Scripts' folder contains all the bash scripts, lists, notes etc. Basically anything written, though mainly scripts are contained in this folder. It is from this folder that a certain testing tool can be configured and run for any application.

The 'Subjects' folder has to contain the applications to be tested. As it is not a good idea to keep binaries and builds in a version control system, the 'android_apps_benchmark' folder only contains the source code of the applications and all these applications are copied in the 'subjects' folder for the purpose of building and testing. This step was taken to ensure the high quality standards of this research.

In addition to these changes, the new versions of software were also configured because the new research uses the up to date versions of these software. Prominent among these were the Google APIs, Apache Ant and Android SDKs.
Once all the setup is done, any selected applications can be tested in the following way:

- ➔ Browse to the scripts folder
- ➔ Modify the file subjects.txt to add the names of the targeted applications
- ➔ Run the command: bash –x run_[Testing tool].sh

For example, if the testing tool selected is Monkey, the command will be run as following:

- ➔ bash –x run_monkey.sh

This will start the emulator and then starts building the application using Apache Ant. Android applications are normally built using either Ant or Gradle. For this research, Ant was selected because of its popularity among developers. Although, Gradle is slowly taking over, due to the newly introduced Android Studio[x].
Once the application is built, the script proceeds to install the *.apk* in the emulator. Then once this installation is complete, the testing starts. Changing the script can do the time allocation for this testing.
Once the testing is finished, the script closes the emulator and stores the results in the following address:

/vagrant/results/[machine name]/[Tool name]/[Application name]/

For instance, if the running machine is run1, and the testing tool is monkey with an application named Facebook, the results will be stored in the following location:

/vagrant/results/run1/monkey/Facebook

These results contain four files containing different information. These files are:

1. Build.log
2. Install.log
3. [Tool name].log
4. [Tool name].logcat

Build.log contains information about the build process. Whether the build was successful or not and any exceptions or errors originated during build process.
Install.log has the installation information of the application in to the emulator. The *.apk* file that is generated by the build process is installed in the emulator and the log for successful or unsuccessful installation is kept in this file.
The log for the tool is a record of all the test cases and the values generated. All the pointer values included there input arguments are stored in this log. Basically this file contains all the information about the actions of the testing tool while testing the application.
Finally the logcat file contains the log information of the application. Unlike mainstream Java applications, Android has its own logging system called Logcat. It allows the developers to log anything by categorizing them with any header. For instance, Log.d represents the debug information and in the log output, if anything contains d as a header, it means this is a debug output. This file hence, keeps track of all the logcat output during the testing of the application.
Once the testing is done, these outputs can be analyzed to understand the working of these testing tools.

4. Benchmark Applications

This section of the thesis discussed the benchmark applications, which were developed for the sole purpose of this research. As mentioned earlier, every application targets a

specific attribute of Android development. Hence, first these attributes will be discussed and then how the corresponding application tackles and incorporates this attribute. What should be kept under consideration is that this research is focused on exploring this technique for further detailed researches in the same way. The long-term goal can be to create a range of benchmark applications that cover the full spectrum of Android development. The number of applications in this benchmark can be somewhere around 100-200. But this research focuses on only 9 attributes to cover the basic grounds for various types of development attributes like UI elements, system events and playbacks. For instance, not all the system events are covered. In the long-term research, all the system events can be covered.
The attributes decided for this research are as follows:

1. Activity Life cycle.
2. Audio Playback.
3. Video Playback.
4. Usage of Camera.
5. Sending SMS.
6. Receiving SMS (Listening of a System Event).
7. Usage of Sensors (Accelerometer chosen for this. Also listens to a System Event).
8. Sending the User to another Application (To Google Maps).
9. Listening to the Change in Network Connectivity (Another System Event).

Before moving further with the explanation of the use of these attributes in applications, it is important to discuss the evaluation criteria for the testing purposes. In the previous research, analyzing the stack output did the coverage evaluation. Although efficient, this process was a bit ambiguous. Hence, it was decided to use Logcat output to analyze the results. All along the applications, logs were created to check if the tool executes certain functionality and what test cases are generated. For instance, while testing the application that uses the camera, does the tool generates a test case that

actually takes a picture successfully? And if it does, can it store it successfully in the memory? This not only made things more clearer, but also more human understandable. But it increased the effort since all these logs have to be checked manually in the Logcat outputs from the tool. And these Logcat outputs tend to be very long for even the shorter periods of running of the application.
All of these attributes were successfully transformed into individual applications. This phase of the research took a major bulk of time and effort of the overall research. First the learning phase of Android development environment, which included a huge amount of tutorials and documentation, followed by the beginning of the development phase. Each of the attributes was carefully incorporated into a working application. The tiny details were taken care of since the application had to be built and run on a virtual machine. Once developed, every application was configured to be able to build with Apache Ant. This included the creation of targets in *build.xml* file and *local.properties* file for declaring the proper SDK and NDK paths of the virtual machine. Once all this process was done and the application was locally tested to make sure the smooth building and running, only then it would be transferred to the version control system. From here, the application would be successfully pulled into the virtual machine. The highlights for these applications are discussed below.

1. **ActivityLifecycle**: This application implements all the functions provided by Google API for the Activity.java class. Any class that is inheriting from Activity, can implement these functions such as *onPause(), onResume(), onStart()* etc. What was done for the purpose of this research was that logs were planted in the implementation of these functions so that when the tool tests these functions; it leaves a track in the Logcat.

2. **PlayAudio**: This application is focused on audio playback using the Android playback API. In addition it also tests some of the UI elements of Android. It

provides an activity with two separate buttons. Each of these when pressed, play an audio file.

3. **PlayVideo**: This application is similar to PlayAudio but focuses on the video playback. When the app begins, it starts playing the video by default. But it can also be played through a button. Hence, it allows the tool to explore two different playbacks, default and voluntary.

4. **CamTestActivity**: This application focuses on using the camera of a smartphone. When started, the application shows a camera interface that allows the user to click anywhere on the screen in order to successfully capture an image. Once captured, this image is stored in the memory and is visible in the gallery. The critical thing to test here is whether the testing tool generates a test case that takes a picture successfully.

5. **SendSMS**: This application focuses on the sending of an SMS using Android API for SMSs. When started the application presents a user interface with a text field, a number field and a send button. User can write anything in the text field along with a phone number in the number field, and press the send button to successfully send this SMS to the specified number. Testing tools will be tested on the basis of test case generation for not only successfully sending a text message, but also for non-successful attempt where the phone number is not valid.

6. **ReceiveSMS**: This application is focused to listen the system event generated by Android API when an SMS is received. If listened successfully, the message is received and displayed in an inbox. This is a critical application since most of the testing tools break when testing the system events. The user interface for this application is a regular inbox, which displays the sender's phone details along

with the text of the message. Any message can be clicked to view the details of the message.

7. **AndroidAcceleration**: This app is created to target the usage of sensors of the smartphones. Although, Android smartphones provide the support for multiple sensors, Accelerometer was chosen for this research because of its wide use across the applications. The application displays two labels representing the X and Y movements. As soon as there is a change in position, the application listens to the system event generated by this change and rewrites these labels to shoe the new X and Y. The testing tools will be tested on the criteria whether a test case is successfully generated to change the position successfully.

8. **SendingToAnotherApp**: This application, as evident from the name, sends the user to another application by creating an Intent (new activities need an intent to get started) and sending this intent to the other application to start an activity of that application. In this application, the Intent created is for Google Maps. A user interface is provided for entering an address, which is then opened using Google Maps. Logs are inserted to tests whether a test case is successful in generating a valid address and open it with Google Maps.

9. **ConnectivityListener**: This application focuses on the listening to change in Network Connectivity of the device. Whenever the device switches between various networks or merely gets connected or disconnected from a specific network, the Android API generates a system event. This application works as a service without any activities to listen to this system event. The tools will be tested on the criteria whether they can successfully generate a test case, which changes the network connectivity of the device.

5. Overview of Testing Tools

This section of the thesis contains some comprehensive information about the existing testing tools and their strategies to tackle the verification process. The purpose of providing this is to make the thesis self-contained. It is important that the reader understands this information before moving further with the thesis. The later sections of this thesis use this information to reason for the decisions taken and paths chosen. Currently, within the Android domain, there are multiple tools for test input generation available, which allow for the detection of errors existing within those applications which are made using the Android technology. However, while it may seem that such tools would only benefit the developers of such applications, by means of reducing the number of errors within their product, they can prove beneficial for those maintaining the Android market, as well as the end users.[xi] Native Code, reflection and code obfuscation are common features found within the apps made on the Android platform, and it is the presence of such elements that results in further testing – quite comprehensively - the limitations of static analysis tools.[xii] It is for these purposes, that is, for the testing of apps made using the Android technology, as well as for evaluating the testing limitations of the tools, dynamic analysis is used in order to ascertain sufficient amounts of data necessary for portraying the behavior of the apps, for analytical purposes[xiii] - Google uses its cloud infrastructure to simulate all the apps, which aids in depicting how the said apps would run for the user, simultaneously citing any malicious behavior.[xiv] It is only after passing this string of tests, that apps are placed on Google Market, and there, further categorized in terms of their various features (such as network usage)[xv] and potential threats (such as potential of security leaks)[xvi] for the benefit of the end user.

I. Random Exploration Strategy

One category of test input generation tools undertakes a random strategy for the purpose of generating input for the applications made using the Android platform. This

is an inefficient strategy as it only generates UI events,[xvii] not only due to the sheer volume of such events, but also because of the fact that only a few of these events would generate a reaction in the application, and that too within a very particular set of conditions.

Many tools employing the random strategy generate random values for intents, whereby which they aim at testing the inter-application communications.[xviii] In order to further enhance the understanding of the random exploration strategy, some of the tools employing this technology have been discussed below:

1. **Monkey:**[xix] This tool is a part of the Android developer's toolkit, and hence is also the most frequently used tool as well. While it does not require any additional installation, at the same time in only utilizes a rudimentary random strategy: it only produces UI events and also treats the application being tested as "black-box".[xx] The user (or developer) sets the number of events they want Monkey to generate; once that number is achieved, Monkey stops, regardless of the outcome.

2. **Dynodroid:**[xxi] Although this tool also employs the random strategy, it provides features above and beyond what Monkey provides. For instance, it has the capability of generating system events relevant to the application. This is done by the using the information obtained from monitoring when a listener of the application is registered within the Android platform, for which it instruments the framework.[xxii] This additional feature makes it more efficient than Monkey.

   Furthermore, Dynodroid has the ability to select Frequency strategy, that is, selecting the least-frequently selected events, while using Biased Random strategy, that is, by factoring in the context as well. This means that the relevant events will be selected more often. This paper also uses biased random strategy.

Another additive feature of the Dynodroid is that it allows the user to manually provide inputs when the exploration is stalling.

3. **DroidFuzzer:**[xxiii] Unlike tools which generate UI events, the DroidFuzzer only generates inputs for activities accepting MIME data types, for instance AVI, MP3, HTML, and so on. Although the DroidFuzzer is to be implemented as an Android application, it remains unavailable.

Random test input generators are efficient at producing events and are therefore also suited well to performing tasks such as stress tests. However, they lack the capability of generating specialized events. Furthermore, they cannot determine the parts of the application which have already been covered, and therefore may end up generating events which would be superfluous and thereby do not aid in the exploration. Additionally, they have a manually prescribed time-out, which has no correlation with the results of the exploration.[xxiv]

II. Model-Based Exploration Strategy

Subsequent to the precedent set by various GUI testing tools for stand-alone applications,[xxv] and that set by Web Crawlers,[xxvi] some of the testing tools developed for the Android platform generate events using the GUI model of application, thereby exploring the application being tested in a systematic manner.[xxvii] "These models are usually finite state machines that have activities as states and events as transitions".[xxviii] A certain sub-set of the tools falling within this category aim at formulating models with greater precision by means of differentiating the state of activity elements when representing states, for instance, the same activity before and after initiating a fragment might get treated as two different states. [xxix]Generally, the tools within these categories are dynamic in nature, and the testing culminates at the point where all the events

generated by the discovered states lead to explored states. Some of these tools are discussed below:

1. **GUIRipper**[xxx] (later became **MobiGUITAR**)[xxxi]: This constructs a model of the app being tested in a dynamic fashion, "by crawling it from a starting state". It maintains a list of events that can be generated in that state of activity, and triggers them methodically. Employing a DFS strategy, when GUIRipper cannot identify any additional or new states, it restarts the exploration from the starting state. However, it cannot highlight such behavior of the app, which depends on system events, as it can only generate UI events. Nevertheless, GUIRipper does have some exclusive characteristics, as compared to other model-based testing tools. For instance, the user has the option of initiating exploration of the application from different starting points, albeit this function can only be performed manually.[xxxii] Furthermore, the testers may also provide a set of input values for the purposes of the exploration. While GUIRipper is a publically available tool, however, it does come with a pair of limitations: it is not open-source, and can only be used on machines using the Windows operating system.

2. **ORBIT:**[xxxiii] It is considered more efficient than GUIRipper, because even though the two use the same exploration strategy, ORBIT also statically analyzes the application's code to discern the UI events in terms of their relevance to specific activities. However, ORBIT is currently unavailable on account of being the property of Fujitsu Labs.

3. **PUMA:**[xxxiv] It is a unique tool, which uses random strategy employed by Monkey (see above). However, its peculiarity stems from design and not from the exploration strategy it uses. Using the basic Monkey random strategy, PUMA has the ability of undertaking any dynamic analysis of applications, which have been made using the Android technology. It also represents the application in terms of

a finite state machine, which renders the implementation of various exploration strategies in a more easy fashion. PUMA also allows the tester to redefine the state generation and logic for generating events. Fortunately, PUMA is open source and also available publically, albeit the compatibility of this tool is limited to the Android technology's most recent release.

4. **SwiftHand:**[xxxv] It uses a dynamic finite state machine model of the application being tested, much like the tools discussed above. The primary objective of this tool is to obtain the widest coverage of the application, which it is being used to test. In order to prevent the application from restarting while crawling, it optimizes the exploration strategy. In terms of the generation of events, SwiftHand does not possess the ability to produce system events, and can only generate touching and scrolling UI events.

5. **A ^{3}E -Depth-first:**[xxxvi] unlike the two tools mentioned above, this is an open source tool. It employs two categorical yet complementary strategies. One, on the dynamic model of the application being tested, it uses depth first search, whereby it replicates the exploration strategy used by the other tools discussed. However, without paying heed to the different states of the elements of the activity, it represents each activity as an individual state, thereby using a form of model representation which is more abstract than that used by other tools. However, this abstraction hinders the tool in identifying certain distinct states, thereby omitting to identify certain types of behavior, which more efficient tools would succeed in analyzing.[xxxvii]

Contrary to the effect of employing a random testing strategy, using a model of the application being tested would curtail the amount of superfluous and unnecessary inputs generated, therefore the model strategy ought to generate more effective results with respect to coverage of the code. However, the state representation of the testing

tools falling within this category act as a constraint.[xxxviii] This is because only very specific event triggers changes in the GUI, which can result in the representation of new states. Therefore, in certain circumstances, then it is possible for the tool to miss certain changes, consider the event irrelevant and consequently proceed to exploring other domains of the application. It can happen when an application is only dealing with system calls and has a little or no UI element.

III. Systemic Exploration Strategy

In order to overcome the limitations of the model strategy discussed above, application testing tools have been developed with the aim of systematically testing the domains within the applications which the other tools discussed fail to. Using systematic exploration strategies can prove to be highly advantageous for the purposes of exploring such behavior in applications, which would otherwise remain undetected by other testing tools, especially those employing random techniques. Some of tools employing these systematic exploration strategies are discussed below:

1. **A ^{3}E-Targetted:**[xxxix] it uses taint analysis to construct the Static Transition Activity Graph of the application being tested. This is an alternative to the depth-first search exploration discussed earlier. This tool generates intents, which makes it more efficient at covering the activities of the application, as compared to the depth-first tool.

2. **ACTEve:**[xl]this is a concrete and symbolic testing tool. Beginning from the point where they are generated, it initiates the symbolic tracking of the events from there, leading to the point where they are handled in the application being tested. Additionally, this tool handles UI events as well as system events.

3. **EvoDroid:**[xli] this tool employs evolutionary algorithms, which maximizes coverage of the application by representing individuals as sequences of test inputs and accordingly implements the input function. Thus it uses the said algorithm for the purposes of generating inputs relevant to the testing. However, this tool is not available publically anymore.

6. Decided Tools

This section focuses on the testing tools selected for this research. Since the idea is to develop a benchmark, all the existing testing tools should be evaluated with a greater number of applications. But due to time and resource limitations, only three testing tools were chosen for this research. The idea was to cover as much ground as possible. For that, the selection of the final tools was done with certain criteria in mind. The biggest input factor for this was the previously done research. The outputs of that research[xlii] were used to focus on deciding the boundary limits of the efficiency of these tools. The second factor was to cover the various testing techniques as much as possible. And the third factor was to incorporate any special processes that can prove vital for the testing procedures. The last but not the least factor was the availability of the testing tool for the public distribution.
Keeping this in consideration, the following tools were finalized:

1. **Monkey**: Monkey was found very efficient in the previous research. Despite being designed on random exploration technique, it was covering most of the applications and finding most faults. Hence, it was included not only for the coverage of one of the techniques but mainly because of its performance in the previous research and also its popularity.

2. **A3E**: The reason behind the inclusion of this tool was also twofold. A3E uses both model based exploration strategy by using depth first search and it also uses systemic exploration technology when performing targeted approach. But in the

end the A3E-targeted could not be used because of its unavailability. So only A3E-depth first was used. Also it served as the lower limit regarding performance as its performance was judged to be in the lower ranks during the previous evaluation.

3. **DynoDroid**: The inclusion of DynoDroid was due to the special features reasoning. Despite also being based on random exploration technique, DynoDroid[xliii] claims to cater the testing of system events. One of the highlights of this thesis is also to try to verify this claim by evaluating it over the benchmark applications that also include applications focusing on system events.

7. Evaluations

This part of the thesis will discuss the experimentation carried out and the analyzing of the results obtained from this experimentation. The efficiency of these testing tools over the benchmark applications will be discussed in detail along with any limitations or exceptional behavior presented by any testing tool.

With everything setup and configured for the experimentation, the testing tools were run one by one on each of the developed applications. The AndroTest environment provides the opportunity to run multiple virtual machines at the same time but one of these virtual machines needs around 4-5 GB of ram and around 30 GB of free hard drive space. Since there was only one resource working on the project and with no other available hardware, this approach was not used and all of the experiments were run on one computer using only a single virtual machine.

Once all the experiments were finished, the results were compiled in one single place for the purpose of evaluation. First of all it was made sure that all the applications were successfully built and installed for every testing tool. Checking all the 27 build and install logs did this. Then, one by one, every application's results were checked to find the printed logs and any other exceptions. These results were, as mentioned earlier, stored

in the Logcat files of each application for each testing tool. Using the *grep* command for Linux did the lookup for printed logs. But the exceptions and crashes were looked up through manual reading and analyzing of the logs. For instance, if PlayAudio application is being evaluated for finding if the audio was played or not by Monkey, the following command was run to check if the inserted log was printed out or not. Assuming that the current directory is */vagrant/results/run1/monkey*.

➔ grep “Playing Audio…” /PlayAudio/monkey.logcat

The text “Playing Audio…” is what was inserted in the appropriate place of the code for this application. The image below shows the corresponding code for the above example.

Figure 4: The Main Activity showing the inserted Log

In this image one can notice that when the MediaPlayer will create and play an audio, the Logcat output will print this message. Hence, if this message can be found in the resulted logs, it means there was a test case, which ran this piece of code at least once. Now the results for each of the individual applications will be discussed for each of the three testing tools.

1. **ActivityLifecycle**: This was the simplest of all of the applications and it was predicted that every tool should easily cover all of the activity lifecycle methods. But even if all of the testing tools produced test cases for some of these methods, none of the tools tested all of them. The methods implemented were:

   onCreate()
   onPause()
   onResume()
   onStart()
   onStop()

   Since *onCreate()* was a trivial method and has to be run for starting the activity, the testing was evaluated for only the rest of the methods.
   The methods covered by Monkey were *onStart()* and *onResume()*. The methods covered by DynoDroid were the same. But A3E covered these two and also *onPause().* Although none of the tools were able to cover all the lifecycle methods, they attempted on this attribute and hence, it was decided to assign a yes to all three of the tools for this attribute.

2. **PlayAudio**: As mentioned earlier, the aim of this application was to test the audio playback and as well as some UI elements of android API. For this application, all the testing tools created appropriate test cases and were able to

play the video successfully. Hence, all three tools received a yes for this attribute.

3. **PlayVideo**: This video had two options for the video playback. One by default, and second was by pressing the button. While Monkey succeeded in doing both, Dynodroid and A3E were only able to play the default mode. And since the default mode is mandatory to start the activity, only Monkey was assigned a yes for this attribute.

4. **CamTestActivity**: This application was to determine if the testing tools could produce a test case that would use the camera and capture an image successfully. Once again, only Monkey was able to do this. DynoDroid and A3E could not capture an image. And hence, only Monkey was assigned a yes for this attribute as well.

5. **SendSMS**: This application was focused on sending an SMS successfully to a given phone number. Logs were inserted for every possible scenario. That is, for successfully sending an SMS, for invalid phone number, etc. But none of the testing tools covered any of these functionalities. Not a single tool was able to create a test case for even trying to send an SMS. Hence, all three of the tools received a no for this attribute.

6. **ReceiveSMS**: This was the first application with a system event. Hence, it was very important to know what happens in this case. Especially from the perspective of DynoDroid, which claims to cater these system events. For this application, Monkey and A3E failed to produce a test case. But as claimed by DynoDroid, it created a successful test case to cover this functionality. It generated a system event for SMS with a dummy phone number and text. This event was successfully listened by the application. While searching for this

success, it was found that an SMS saying “Come over! lets have a coffe” from phone number ‘6789077112’ was received by the Broadcast Listener of the application. The image below shows the presence of this test case. Only Dynodroid was assigned a yes for this attribute.

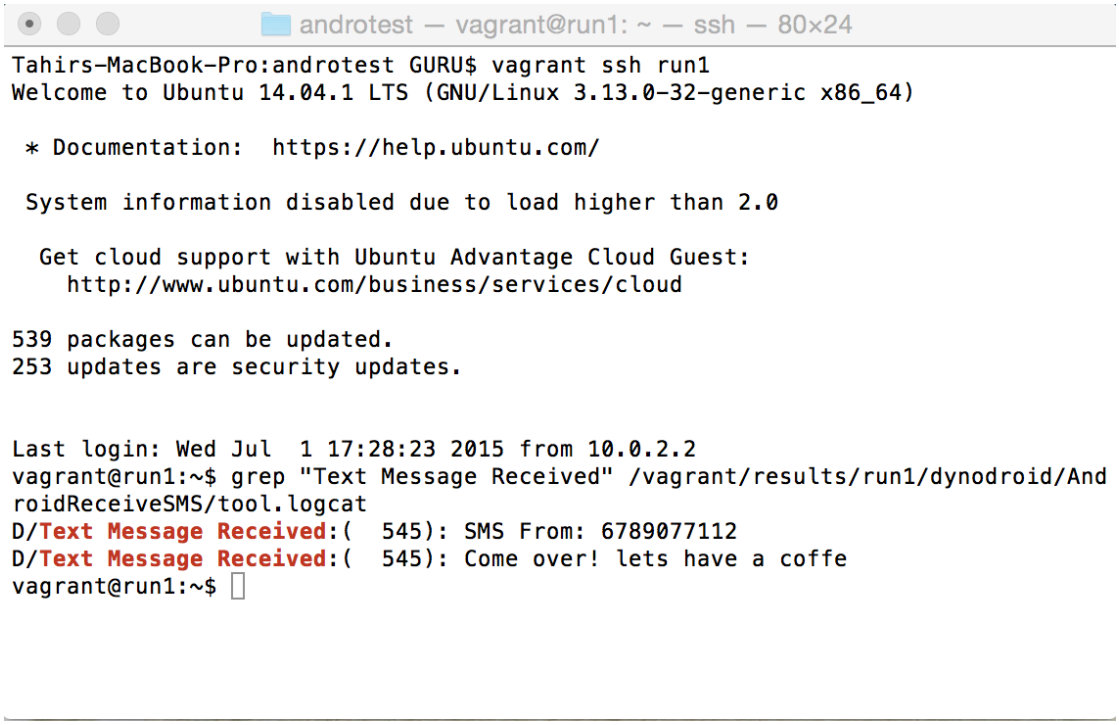


Figure 5: Generation of a System Event by DynoDroid

7. **AndroidAcceleration**: This application was also focused on listening to a system event. And also to check the behavior of testing tools for an application that uses one of the available sensors in an android device. This application was implemented as a service with no activities and hence, no user interface. What happened during the evaluation of this application was somewhat surprising. While none of the tools were able to create a test case for detecting the change in acceleration of the device, Monkey did not even run on this application. When the file Monkey.log (The file containing the information about tool’s running) was analyzed, it was found that Monkey aborted the run as there were no activities found. So, it revealed that Monkey needs activities to test an

application and hence, cannot be used to test the services. All three testing tools were assigned a no for this attribute.

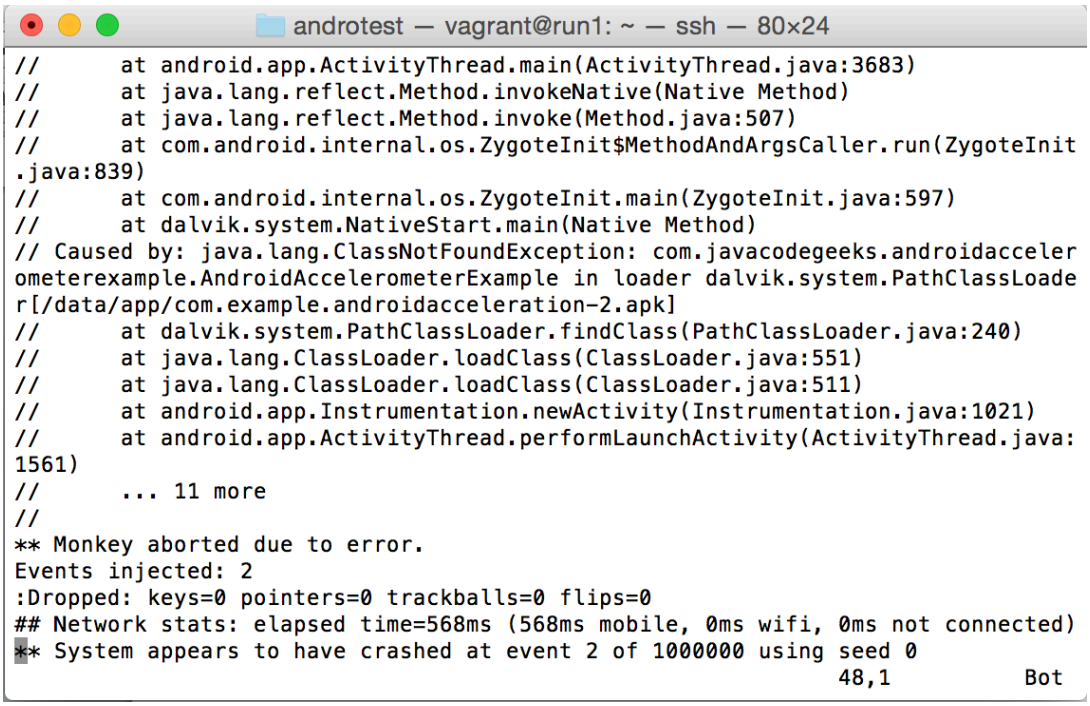


Figure 6: Monkey Aborted for Services

8. **SendingToAnotherApp**: The purpose of this application was to generate an Intent and send this intent to another application to start an activity of this second application. Logs were inserted to check various scenarios. If the second application was started successfully or if the intent was not safe and was not sent to the application. Also if the Address field (Since Google Maps was used as the second application) was left empty. But all three of the testing tools could not cover any of these functionalities. And a no was assigned to all three of the testing tools for this attribute.

9. **ConnectivityListener**: Another system event was targeted by this application for detecting the change in network connectivity. The goal was to see if any testing tool generates a test case, which changes the network connectivity of the device

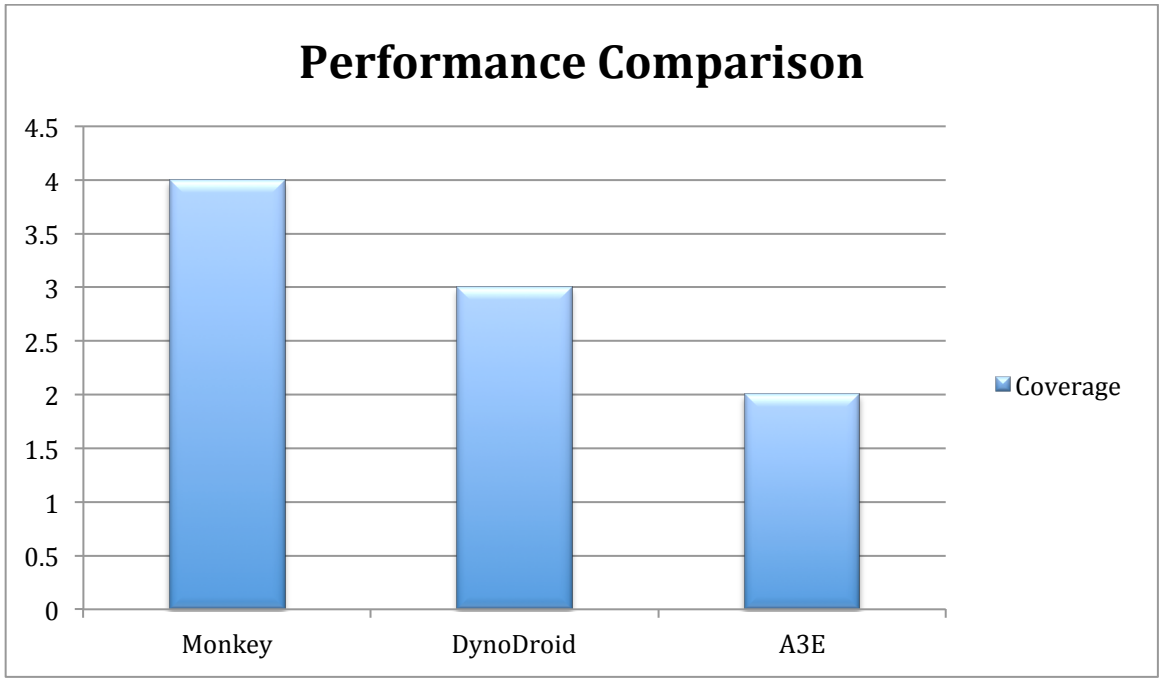
so that it can be listened by the Broadcast Receiver implemented by this application. Again, all three of the testing tools failed to generate such a test case. Even DynoDroid, which succeeded in generating a system event for SMS, failed to do so in this case. Hence, all three of the testing tools were assigned a no.

Compiling the data provided in the above-mentioned details, the following table was created to edify the performance of these testing tools for testing the benchmark applications. The '✔' means the tool was successful in covering the functionality and '✖' means it was not able to do so.

Applications | Tools ➡ ⬇	*Monkey*	*DynoDroid*	*A3E*
PlayVideo	✔	✖	✖
ActivityLifecycle	✔	✔	✔
AndroidAcceleration	✖	✖	✖
CamTestActivity	✔	✖	✖
SendSMS	✖	✖	✖
PlayAudio	✔	✔	✔
AndroidReceiveSMS	✖	✔	✖
ConectivityListener	✖	✖	✖
SendingToAnotherApp	✖	✖	✖

Table: Performance of Testing Tools

To compare the performance of these testing tools over these applications, below is a bar chart that shows the comparison with respect to the coverage.



Monkey is leading with most coverage followed by DynoDroid and then A3E. This provides with the similar result as obtained from the previous researchxliv on the topic. Although the difference is not that much here but for a bigger benchmark, this difference can increase.

8. Conclusion

There are multiple things that can be determined from this research. First of all, it can be seen that this research reiterates its predecessor research by producing similar results. Monkey, despite being based on random exploration technique, performs better than its counterparts. The reason behind it can be that in real world scenarios, the applications follow somewhat similar elements, which Monkey targets randomly. For instance, UI elements of the application, which are very common in almost all the

applications except for services. Whereas more sophisticated techniques try to approach the problem of testing through a systemic way, which targets all the attributes. So this conservative approach of Monkey can be the edge, which makes it more efficient in real applications. But at the same time it is evident that Monkey does not cater services. So for the developers of services, it is useless to run Monkey for their projects.

DynoDroid showed that it better for system events as compared to its counterparts. There were only three applications focused on system events whereas there are a lot more system events like battery shortage, phone calls etc. If all of these system events are targeted, it is possible that DynoDroid will be able to generate the test cases for these system events.

A3E performed rather poorly as compared to its counterparts, but it also showed that systemic depth first approach managed to target the inherited lifecycle functions. It was able to create test cases that paused the application and then restarted them. So for the applications that are used on a regular basis and hence, have a long and busy lifecycle, might benefit from using this tool. One example for such applications is Whatsapp.

Another important thing realized during this research was the ease of use and resource usage by these tools. While Monkey and A3E are easy to run on applications, DynoDroid provided with some difficulty in this regard. It requires very specific emulator configurations and a lot of setup prior to its ready state when it can actually start the testing process.

It was also noted that that DynoDroid and A3E tend to clean up the environment after finishing their run on an application and before starting the next. Although in this research, a fresh emulator was used for every application. But this feature of the tool can be useful and as well as harmful. The useful part is that it avoids the side effects of previous runs to interfere with the next ones. And it is harmful because it erases all the data and doesn't reuse any of the configurations, which can be time and resource consuming.

It is also evident that Android applications follow a different pattern as compared to normal Java applications, which tend to be more responsive for systemic exploration techniques rather than random exploration techniques[xlv].

The major conclusion that is evident from the results is that a further research in this area can be very useful. If a rather small-scale research can unfold these recommendations, a long-term research targeting all the attributes of Android API and all the available testing tools can provide with a lot more useful and interesting revelations.

9. References

[i] "Google shows off new version of Android, announces 1 billion active monthly users". Techspot. Retrieved June 30, 2014

[ii] "Android's Google Play beats App Store with over 1 billion apps, now officially largest". Phonearena.com. Retrieved August 28, 2013.

[iii] Developer Economics Q3 2013 analyst report- http://www.visionmobile.com/DevEcon3Q13 – retrieved July 2013

[iv] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[v] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[vi] Understand the Lifecycle Callbacks,

http://developer.android.com/training/basics/activity-lifecycle/starting.html.

Retrieved May 27th, 2015.

[vii] C. Hu and I. Neamtiu. Automating GUI Testing for Android Applications. In Proceedings of the 6th International Workshop on Automation of Software Test, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[viii] Download Virtual Box, https://www.virtualbox.org/wiki/Downloads. Retrieved June 5th, 2015.

[ix] Download Vagrant, https://www.vagrantup.com/downloads.html. Retrieved June 5th, 2015.

[x] Android Builds evolved, with Gradle, http://developer.android.com/training/index.html. Retrieved June 7th, 2015.

[xi] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xii] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM; A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 1025–1035, New York, NY, USA, June 2014. ACM.

[xiii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xiv] H. Lockheimer. Google bouncer. http://googlemobile.blogspot.com.es/2012/02/android-and-security.html.

[xv] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer Profiling of Android Applications. In Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Mobicom '12, pages 137–148, New York, NY, USA, 2012. ACM.

[xvi] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[xvii]S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.1 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xviii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.1 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xix] The Monkey UI android testing tool. http://developer. android.com/tools/help/monkey.html.

[xx] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.1 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxi] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[xxii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.1 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxiii] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In Proceedings of International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.

[xxiv] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.1 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxv] F. Gross, G. Fraser, and A. Zeller. EXSYST: Search-based GUI Testing. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 1423–1426, Piscataway, NJ, USA, 2012. IEEE Press; L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 81–90, Washington, DC, USA, 2012. IEEE Computer Society; A. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of the 10th Working Conference on Reverse Engineering, WCRE '03, pages 260–, Washington, DC, USA, 2003. IEEE Computer Society.

[xxvi] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. WebMate: Generating Test Cases for Web 2.0. In Software Quality. Increasing Value in Software and Systems Development, pages 55–69. Springer, 2013; A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. ACM Transactions on the Web (TWEB), 6(1):3:1–3:30, 2012; S. Roy Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 702–711, Piscataway, NJ, USA, 2013. IEEE Press.

[xxvii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxviii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxix] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxx] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In Proceedings of

the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[xxxi] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR – a tool for automated model-based testing of mobile apps. IEEE Software, PP(99):NN–NN, 2014.

[xxxii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxxiii] W. Yang, M. R. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[xxxiv] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[xxxv] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[xxxvi] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[xxxvii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxxviii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.2 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xxxix] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[xl] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[xli] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented Evolutionary Testing of Android Apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, New York, NY, USA, 2014. ACM.

[xlii] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.3 <http://arxiv.org/abs/1503.07217>. 31 March 2015.

[xliii] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[xliv] S. Roy Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? Available at: Cornell University Library, Section 3.3 <http://arxiv.org/abs/1503.07217>. 31 March 2015 .

[xlv] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 81–90, Washington, DC, USA, 2012. IEEE Computer Society.